CSE-276C HW4

Zhenyu Wu | PID: A69030822

December 7, 2024

1 Question 1: C-Space

I applied the binary occupancy grid in MATLAB to represent the map. To build c-space, I approximate robot's radius by considering the robot's orientation (angle) and side length. Specifically, I compute the projection of half the side length onto the x and y axes. Then I sum both projections to make a conservative estimate of the maximum extent of the robot in any direction, to ensure sufficient clearance in the c-space. Based on this method, when the orientation is at 45, 135, 225 and 315, it would have the max radius and inflation to the boundary and obstacles, which equal to the half-diagonal. On the other hand, when the orientation is at 0, 90, 180, 270, 360, it would have the min radius and inflation which equal to the half side length. Below are the illustration of c-space at 0, 45 and 90 degree of orientation.(Please ignore the unit label (meters) in x and y axis, the actual unit is ft)

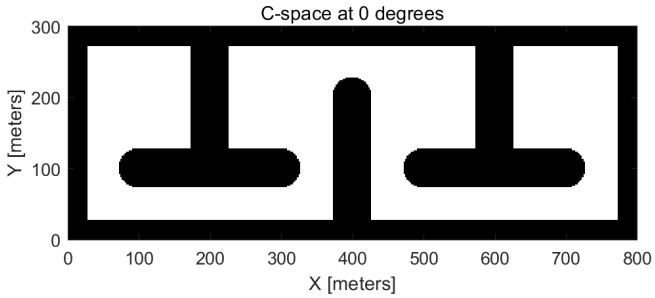

Figure 1: C-Space: 0 degree

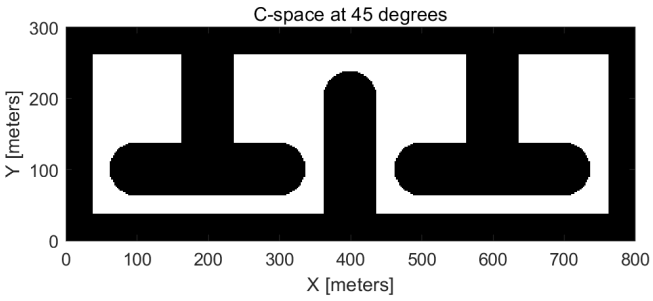

Figure 2: C-Space: 45 degree

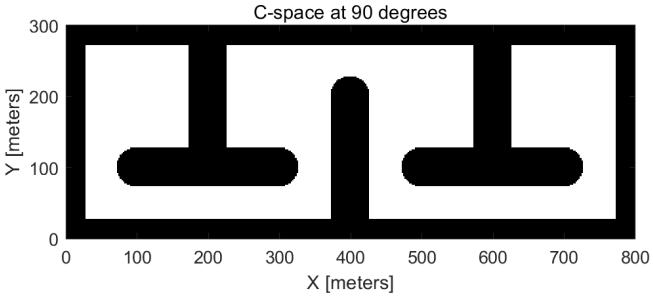

Figure 3: C-Space: 90 degree

Then I exported the occupancy matrix of each c-space at different orientations (totally $360/5 = 72$) to do the planning. The map resolution is 0.5 (each grid is 2x2 in size), the matrix is size 150x400. This reduction reflects the rescaling of both the map and the obstacles into 2-ft units. For example, the start

and goal will be (25, 25) and (375, 25). **The illustration of the following session are made on this occupancy matrix directly.**

2 Orientation Planning

For all the following path and road map illustration, my implementation provides this guarantee: **If there is a edge between two nodes** (x, y, θ) **and** (x', y', θ')**, robot can rotate from** θ **to** θ' **at** (x, y) **and move along the line connecting** $(x, y), (x', y')$ **with** θ' **orientation (every grids on this line in c-space of orientation** θ' **are free).**

3 Question 2: Greedy Search

For this question, I directly implemented and applied a A* search algorithm to do the greedy search on the occupancy matrix with the general framework as below:

Algorithm 1 A* Pathfinding Algorithm

Input: Start node and goal node.
Output: Path from start to goal, or failure if no path exists.

```
Initialize open_list with the start node  Initialize closed_set as an empty set
while open_list is not empty do
    current_node ← node with the lowest f in open_list  Remove current_node from open_list
    if current_node is the goal then
        Reconstruct path and return
    Add current_node to closed_set
    foreach neighbor of current_node do
        if neighbor is in closed_set then
            continue
        tentative_g ← current_node.g+movement_cost(current_node, neighbor)
        if neighbor is in open_list and tentative_g < neighbor.g then
            Update neighbor.g, neighbor.f, and set neighbor.parent ← current_node
        else if neighbor is not in open_list then
            Compute neighbor.g, neighbor.h, neighbor.f  Set neighbor.parent ← current_node  Add neighbor to open_list
return Failure (no path found)
```

This is a commonly used general framework for the A* algorithm. However, the key modifications I made to implement a true 3D search lie in the specific

computation of the cost-to-come(g), the heuristic (h), and the process for generating valid neighbors. These components were adapted to account for the additional dimension and the constraints unique to the 3D environment.

3.1 Movement Cost (g)

To compute the movement cost, I treat a movement across one grid as 1 step and rotation of each 5 degree as 1 step. For instance, if the robot want to move from (0,0,0) to (0,3,90), then the movement cost (g) will be Translation Cost + Rotation Cost $= 3 + 90/5 = 3 + 18 = 21$. Also this logic is based on the idea that the robot should always face the direction will it will go. For instance if the robot want to move from (0,0,0) to (0,3,90), then the robot will first rotate to 90 degree and then move. In the computation for generating valid neighbors, I enforced this constraint.

$g(node, target)$ = Translation Cost per grid+Rotation Cost per angular resolution

3.2 Heuristic (h)

To compute the heuristic, I applied the **Euclidean Distance** to calculate the position difference to the goal and combined it with the orientation difference to the goal, so

$$h(node) = \sqrt{(x - x_{\text{goal}})^2 + (y - y_{\text{goal}})^2} + \frac{\min(|\theta - \theta_{\text{goal}}|, 360 - |\theta - \theta_{\text{goal}}|)}{\text{angular\_resolution}}$$

For all 4 planning, they all use Euclidean Distance in Heuristic.

3.3 Compute Neighbor

The general process of computing neighbor is

1 Generate possible neighbor

2 Check if neighbors are occupied

3 Check if robot could rotate to face the neighbors (5 degree per time)

For generate possible neighbor, I restrict that the robot would only have 8 neighbors, which includes

1 upward (0,1,90)

2 downward (0,-1,90)

3 left (0, -1, 180)

4 right (0, 1, 0)

5 up-left (-1, 1, 135)

6 up-right (1, 1, 45)

7 down-left (-1, -1, 225)

8 down-right (1,-1,315)

Then I check the occupancy matrix to see if a neighbor is blocked or occupied. Finally, as I mentioned, I constrained the robot to always face the direction it will move. The robot attempts to rotate to face each neighbor in **5-degree increments**. If any rotation during this process is not achievable (e.g., due to collision or constraints), then this neighbor will be discarded from the list of valid moves.

3.4 Result

Below is the result of the greedy search; I added arrows for each grid to represent the orientation of robot on the path. I run the search multiple times, and they all generate the same solution.

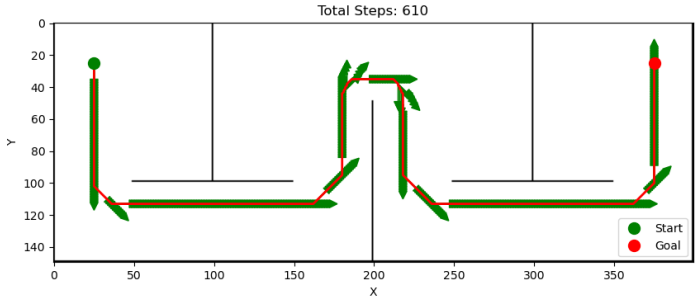

Figure 4: Path: Greedy Search

For the length of the path, I measured **how many grid the path traversed**, for the greedy search, the length is 610.

4 Voronoi Diagram Based

For the safety-prioritized path, I applied voronoi-based planning which includes following steps:

1 Generate voronoi diagram

2 Perform A* on voronoi diagram

4.1 Voronoi Diagram

I directly applied the scipy voronoi package to generate the voronoi diagram. Since this question ask us to generate the safest path, **I generate the voronoi diagram in the c-space of 45 degree**, since this orientation would have the largest inflation and most limited space, which means the free-space in this c-space will also be a free-space for all other orientation. Below is the illustration of the diagram (after I cleaned the unreachable edges and vertices)

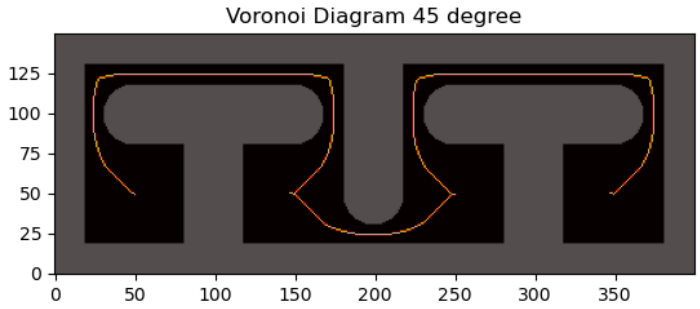

Figure 5: Voronoi Diagram

Since the row and col of occupancy matrix correspond to the y and x value and I forgot to flip this one, it is upside down. But the diagram is correct.

4.2 A* modified

For this question, I modified the neighbor generation logic mentioned before. Now the neighbor would only be selected along the path of the diagram.

4.3 Result

Below is the result of the voronoi diagram; I added arrows for each grid to represent the orientation of robot on the path. I run the search multiple times, and they all generate the same solution.

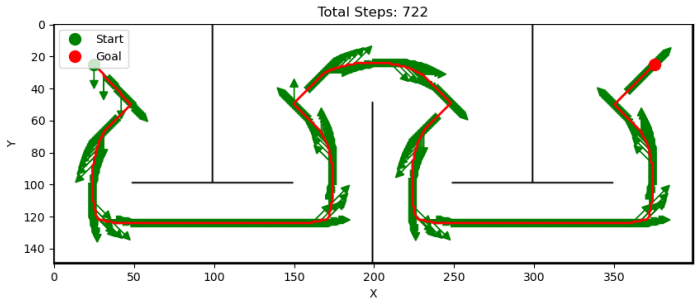

Figure 6: Path: Voronoi Diagram Based

For the length of the path, I measured **how many grid the path traversed**, for the voronoi diagram, the length is 722.

5 PRM

The general workflow of PRM path planning is

1 Generate PRM road map

2 Perform A* on the PRM road map

5.1 PRM Road Map

To generate PRM road map, I applied the conventional algorithm as below

Algorithm 2 PRM

Input: n: number of sample points
k: number of closest neighbors to examine for each configuration
Output: A roadmap $G = (V, E)$

```
V ← ∅  E ← ∅
while |V| < n do
    repeat
        q ← a random configuration in Q
    until q is collision-free;
    V ← V ∪ {q}
foreach q ∈ V do
    N_q ← the k closest neighbors of q chosen from V according to dist  foreach
     q' ∈ N_q do
        if (q, q') ∉ E and Δ(q, q') ≠ NIL then
            E ← E ∪ {(q, q')}
```

5.2 Sampling in 3D

I can think of two strategies for sampling in 3D: 1. Randomly sample x, y and theta; 2. Fix theta and randomly sample x,y. For my implementation, I tried the 1st strategy: sample x, y, and theta. Then perform the rotation check (5 dgree per time) as well as translation check (same as what I mentioned in 3.3) to connect neighbors to provide the guarantee mentioned in section 2. Below are the PRM of 50, 100 and 500 sample points.

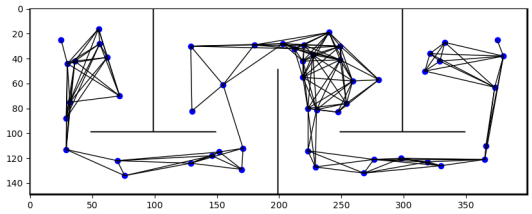

Figure 7: PRM Map: 50 Samples

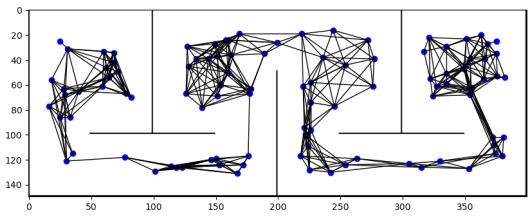

Figure 8: PRM Map: 100 Samples

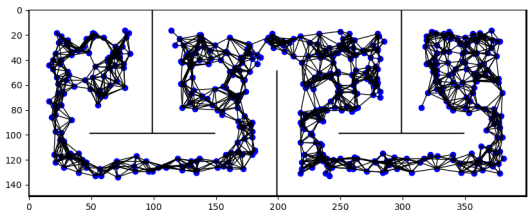

Figure 9: PRM Map: 500 Samples

The Pro of this strategy is that it considered all possible orientations' free-space, which means it might have greater chance of finding shorter path. However, the Con of this strategy is that the orientations of robot at each vertex on this map are not planned but randomly selected. **But the guarantee mentioned in section 2 still holds.** Since on Piazza, we are given that the robot could move in any direction and orientation as long as collision-free, I applied this strategy. For the second strategy, there is a conservative way of doing it

which is only sample (x, y) in c-space of 45 degree (like I mentioned in previous question), then do orientation planning on vertex. The Pro of this method is that it will allow the robot to have a more reasonable orientation along the path. But the Con is that it will not be able to find shorter path.

5.3 Result

Below are the results of the PRM. Unlike previous path, I only added arrow on each vertex. The guarantee mentioned in section 2 still holds.

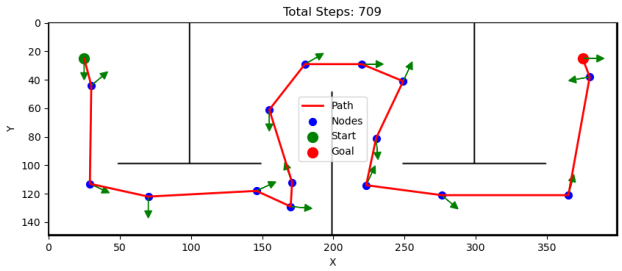

Figure 10: Path: PRM Sample 50

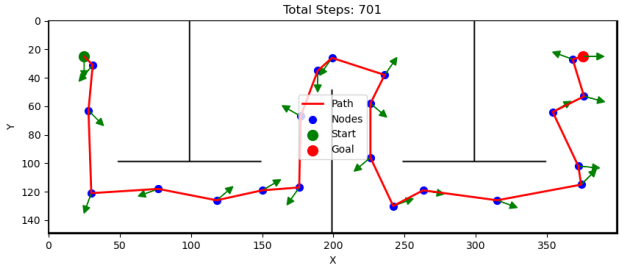

Figure 11: Path: PRM Sample 100

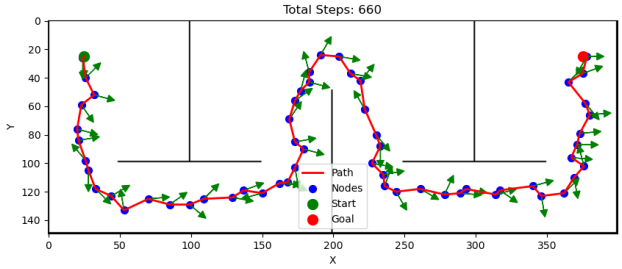

Figure 12: Path: PRM Sample 500

For the length of the path, I measured **how many grid the path traversed**. Based on my test, it is harder to find a shorter or even find a path

from start to goal with lower sample points such as 50. Since the PRM makes edges in neighborhood (select closest k^{th} neighbor, it might generate unconnected diagram and paths with less sample points. More sample point will increase the chance of finding a valid path(connected diagram and path) and shorter path (can find path with length less than 700 only with sample point 500). For the length, 50 sample points with length of 709, 100 with length of 701 and 500 with length of 660.

6 RRT

The overall method is the same as PRM

1 Generate RRT road map

2 Perform A* on the RRT road map

6.1 RRT Road Map

The RRT road map is constructed by

Algorithm 3 RRT Sampling Algorithm

Input: *start*: Start node, *num_sample*: Number of samples, *cspace*: Configuration space

Output: Tree of vertices and edges

Add *start* to *vertices*
while $|vertices| < num\_sample$ **do**
    Randomly select an angle from *angles* Let $sample\_space \leftarrow cspace[angle]$
    Find indices of free space in *sample_space* Randomly select a free space index Let $node \leftarrow (free\_index[0], free\_index[1], angle)$
    if *node already exists in vertices* **then**
        continue
    Find nearest vertex *neighbor* to *node*
    if *there is no collision between neighbor and node* **then**
        Add *node* to *vertices* Let $node\_ind \leftarrow index(node)$ Let $neighbor\_ind \leftarrow index(neighbor)$ Add edge $(neighbor\_ind, node\_ind)$ to *edges*

Like I mentioned in the PRM section, I randomly sampled x, y, θ, so the orientation at each vertex is randomly sampled.
However, it is extremely hard to find a valid path from start to goal with this conventional implementation, since we restrict the number of points to sampled. With this conventional implementation, it should continue sample until reach time t. Unlike PRM, which could make edges with-in neighborhood, if the tree is not expanded toward right enough, every vertex sampled in the right of the map would be discarded. I have below map for this conventional implementation generated at 3 sample rate

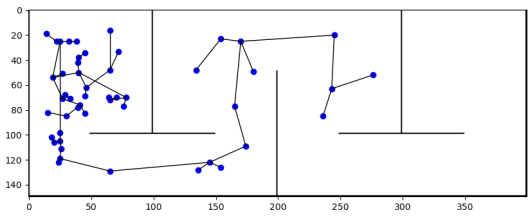

Figure 13: RRT Map: 50 Samples

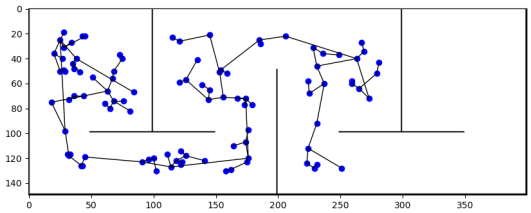

Figure 14: RRT Map: 100 Samples

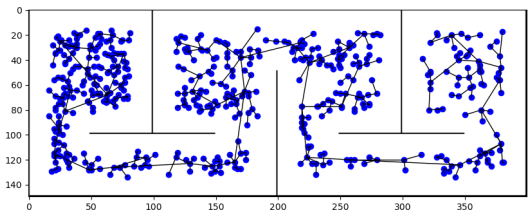

Figure 15: RRT Map: 500 Samples

For sample point of 500, it is possible to find path, but quite impossible for 50 and 100, even after I make them strongly biased while sampling(70 % random sample, 30 % sample in the region with some distance from goal). Then, I decided to use another strategy for 50 and 100 sample point. **But for 500, I applied this conventional RRT without bias**.

6.2 Guided RRT

For 50 and 100 sample point, I added some intermediate goal which would guide the tree to expand toward target, and I also applied some *crazy* bias strategy. For 50 sample point, 50% of chance to sample toward intermediate goals; 30 % of chance to sample toward intermediate goals for 100 sample points. And finally I have below maps

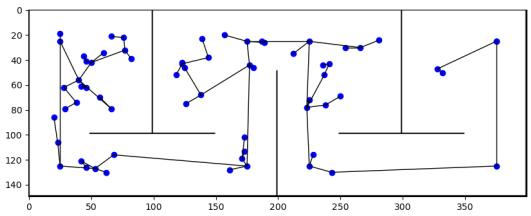

Figure 16: RRT Map Guided: 50 Samples

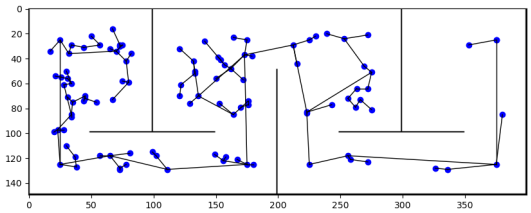

Figure 17: RRT Map Guided: 100 Samples

However, this method I think is not align with the core of RRT. By providing guiding and crazy bias rate, the random sample process becomes quite deterministic.

6.3 Result

Below are the results for RRT. Like I mentioned, 50 and 100 sample points uses the guided RRT, and 500 sample points uses the conventional RRT.

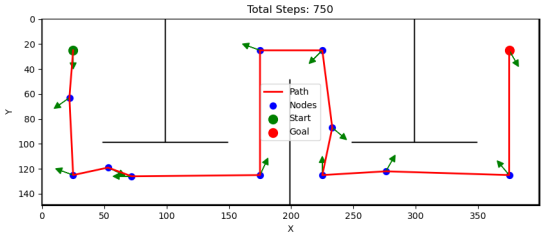

Figure 18: RRT Path: 50 Samples

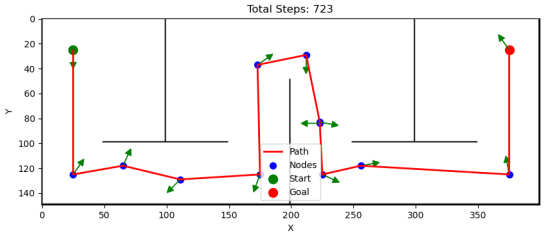

Figure 19: RRT Path: 100 Samples

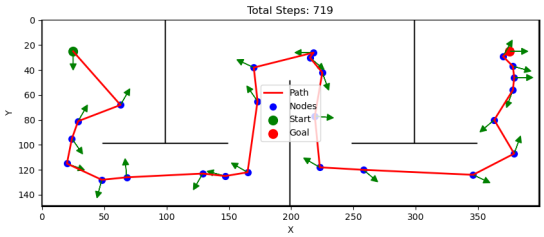

Figure 20: RRT Path: 500 Samples

For the length of the path, I measured **how many grid the path traversed**. With the sample point 50, the length of path is 750; with sample point 100, the length is 723; and the length is 719 with sample point 500.

7 Compare: PRM and RRT

For this path planning across long-horizontal area, PRM could find valid path with small sample points (50 and 100), but RRT can't. The reason is that PRM would connect vertices in local neighborhoods while RRT relies on incremental random sampling to grow the tree. In a long and narrow area, RRT struggles to find a valid path because it requires a series of successful connections to incrementally reach the goal, which is challenging in such constrained spaces. On the other hand, the con of PRM is that it might generate unconnected graph but RRT is guaranteed to grow a connected tree from the start point, ensuring that every node in the tree is reachable from the root. This makes RRT particularly useful in environments where connectivity is difficult to establish or the free space is highly irregular. However, RRT requires more samples and computational effort to cover the space effectively, while PRM can provide a more global view of the environment with fewer samples but risks leaving parts of the space disconnected if the sampling density is insufficient.